Information Systems Frontiers 4:3, 331–342, 2002

Domain Modelling and the Co-Design of Business Rules in the Telecommunication Business Area

Monique Snoeck and Cindy Michiels***
Management Information Systems Group, K.U. Leuven, Naamsestraat 69, B-3000 Leuven, Belgium
E-mail: monique.snoeck@econ.kuleuven.ac.be
E-mail: cindy.michiels@econ.kuleuven.ac.be

Abstract. *This paper discusses the development of an enterprise domain model in an environment where part of the domain knowledge is vague and not yet formalised in company-wide business rules. The domain model was developed for a young company starting in the telecommunications sector. The company relied on a number of stand-alone business support systems and sought for a manner to integrate them. There was opted for the development of an enterprise-wide domain model that had to serve as an integration layer to coordinate the stand-alone applications. A specific feature of the company was that it could build up its information infrastructure form scratch, so that many aspects of its business were still in the process of being defined. The paper will highlight parts of the Enterprise Model where there was a need for co-designing business rules together with the domain model. A result of this whole effort was that the company got more insight into important domain knowledge and developed a common understanding across functional areas of the way of doing business.*

Key Words. *domain modelling, business rules, object-oriented analysis, business process modelling*

1. Introduction

The paper presents an enterprise and business modelling project for a young company starting in the telecommunications area. The company positions itself as a broadband application provider and is specifically tailored towards the SME market. The company relies on a number of stand-alone business and operational support systems (BSS/OSS). A key factor in focusing on the SME market is the ability to handle large volumes of small orders. The company recognises that the integration of its stand-alone support systems can significantly improve the transaction processing volume.

There was opted for the development of an enterprise-wide domain layer, from now on called Enterprise Layer, that will serve as an integration layer to co-ordinate the previously stand-alone business applications. The Enterprise Layer will prevent redundancy in data storage and data processing, thereby improving the transaction processing volume.

A particular feature of working with a young company was that the Enterprise Layer could be build from scratch, there was no need for re-engineering an existing legacy infrastructure. The major advantage were the degrees of freedom in developing the Enterprise Layer, there was no "history" to be taken into account. The major difficulty was that many work procedures were still under construction and had to be figured out during the modelling of the Enterprise Layer. Although the top-level business processes had already been defined, the business rules still remained in the process of being defined.

During the specification phase it became very soon apparent that there was not a well-defined and company-wide understanding of important concepts such as PRODUCT and SALES ORDER. Because of the use of stand-alone BSS/OSS, each functional area had its own definition of domain concepts and business rules. Many of the low-level business processes were not formalised yet. Existing work procedures were often designed as solutions to day-to-day problems, lacking the genericity to scale up to larger transaction volumes and to introduce new product types.

The development of the Enterprise Layer has forced the company into a common understanding across

*To whom correspondence should be addressed.

functional areas of the way of doing business, of important domain knowledge and of business rules that govern business objects and business events.

The rest of the paper will be organised as follows: Section 2 describes the set-up of the project. The integration strategy of the company is summarised in 4 steps. The focus of this paper will be on step 2, the modelling of an Enterprise Layer. The next two sections both illustrate how a specific modelling problem could be solved by designing a number of business rules as part of the Enterprise Layer: Section 3 describes how an integrated view on PRODUCT could be maintained across different business units, Section 4 deals with the managing of sequences of business activities. The diagrams are presented using standard UML notations, but in practice the modelling method MERODE (Snoeck and Dedene, 1998; Snoeck et al., 1999) was used, as this method is specifically tailored to the development of enterprise models. In a modelling environment where a large part of the domain knowledge and business rules are still vague at the start of the project, prototyping can be useful as a technique to gain more confidence in the specified model and chosen options. Section 5 deals with prototyping the Enterprise Layer. To conclude, Section 6 presents a number of insights gained from the project.

2. Enterprise Layer to Co-Ordinate Stand-Alone BSS/OSS

2.1. Integration approach in 4 steps

The automation strategy of the company is summarised by the following four steps:

Step 1: Roll out of industry proven BSS/OSS with out of the box functionality. Each application is treated in isolation (total lack of integration).

Step 2: Specification and development of an Enterprise Layer that will support all applications and user interfaces. The Enterprise layer is by definition passive and not aware of its users.

Step 3: Integration of the existing BSS/OSS by plugging them in on the Enterprise Layer. The interface between these applications and the Enterprise Layer will be realised by designing agents, responsible for co-ordinating the information exchange.

Step 4: Development of user interfaces on top of the BSS/OSS applications or directly on top of the Enterprise Layer.

At the start of the project, step 1 had been realised: three out of the four main functional domains were supported by a standalone software package. The top-level business processes of the company are represented in Fig. 1. There exists no automated support for the sales domain: sales business processes are mainly paper-based or use standard office software such as word processors and spreadsheets. The Service Provisioning domain is supported by the software package TBS Metasolv®,[1] the Billing domain by the package Geneva (now Convergys)[2] and the Customer Support domain by the Clarify software[3] .

The development of the Enterprise Layer and the subsequent steps eventually lead to the layered infrastructure depicted in Fig. 2. The previously stand-alone BSS/OSS constitute now the middle layer. The Enterprise Layer serves as a foundation layer for them. Co-ordination agents realise the information exchange between the business applications and the inherently passive Enterprise Layer. Together, the Enterprise Layer and the co-ordination agents constitute the Integration scope. The top layer is established by the development of user interfaces that offer a Web interface to both the business applications and to parts of the Enterprise Layer.

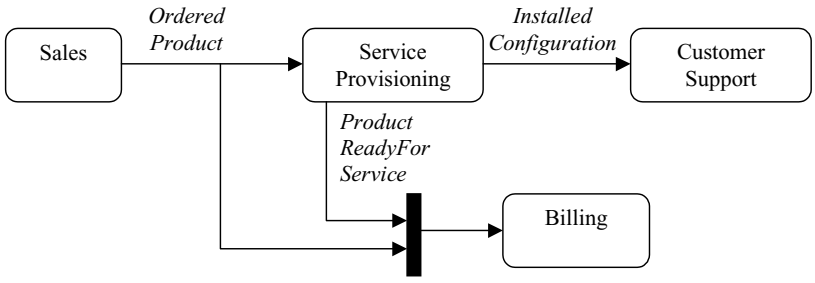

Fig. 1. *Main business process.*

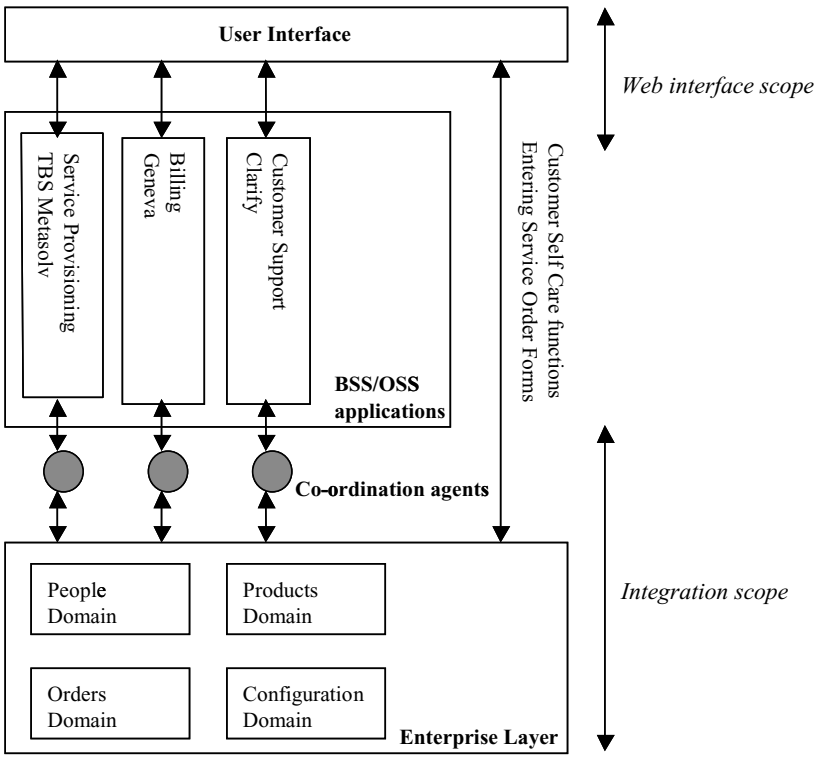

Fig. 2. *Overview of company automation.*

2.2. Step 2—developing an Enterprise Layer

The focus of this paper will be on Step 2 of the automation approach: the definition of an Enterprise Layer. This layer covers the company's four main business domains: People, Products, Orders and Configuration. A short overview of each business domain is presented below.

The *People* domain concerns both the customers and the sales persons. Information about people is found in all four business processes. The sales process stores data on sales people (both in-house and distributors) and on commercial contacts for customers. The Service Provisioning application and the Customer Support application both maintain data on technical contacts. Finally, the Billing application keeps track of data about financial contacts. Since the company mainly deals with SME, a single person often takes several roles simultaneously, so that information about the same person can be distributed and replicated across several business processes. The Enterprise Layer will ensure that information about an individual person is stored and maintained in a single place.

The *Product* domain maintains all information related to products sold by the company. Depending on the business process, a different view on products is needed. Formerly, each BSS/OSS maintained its own product catalogue. The Enterprise Layer will be responsible for tying together the description of products. Therefore, a distinction had to be made between a *commercial* product view and a *technical* product view in the Product Domain.

The *Orders* domain handles the registration of sales orders. A sales order can contain multiple order lines, one for each ordered product. Business rules will guard the mapping between orders in the Order Domain and commercial product descriptions in the Product Domain.

The *Configuration* domain keeps track of the technical configuration that is build at a customer's site during the provisioning activities. It is information that is produced by the Service Provisioning application and is used by the Customer Support application. Business rules will guard the mapping between installation activities in the Configuration Domain

and technical product descriptions in the Product Domain.

The example presented in Section 3, will explain in more detail how both perspectives are maintained in the Enterprise Layer. A solution was found in the adoption of the TypeObject Pattern[4] together with the design of mapping business rules. Together they will give a formal structure to key business concepts such as PRODUCT, INSTALLATION ORDER and PART.

Section 4 will demonstrate how the adoption of company-wide business rules in the Enterprise Layer can formalise existing work procedures. Whereas Section 3 is in essence an example of information structure, Section 4 relates to the behavioural perspective.

2.3. Methodology

The diagrams in Sections 3 and 4 are presented using standard UML notations, but in practice the modelling method MERODE (Snoeck and Dedene, 1998; Snoeck et al., 1999) was used, as this method is specifically tailored to the development of enterprise models. MERODE advocates a clear separation of concerns, in particular a separation between the information systems services and the enterprise model. The information systems services are defined as a layer on top of the enterprise model, what perfectly fits with the set-up of the project.

A second interesting feature of MERODE is that it does not rely on message passing to model interaction between domain object classes. Instead, business events are identified as independent concepts. An object-event table allows defining which types of events affect which types of objects. When an object type is involved in an event, a method is required to implement the effect of the event on instances of this class. Whenever an event actually occurs, it is broadcasted to all involved domain objects. This broadcasting paradigm requires the implementation of an event-handling layer between the information system services and the enterprise layer. This approach allows implementing the Enterprise Layer (together with the event-handling layer) both by means of an object-oriented implementation technology as a relational or object-relational technology. Since not all BSS/OSS were object-oriented, there was still enough freedom to choose the most appropriate implementation technology for the Enterprise Layer, without danger for a paradigm mismatch between specification and implementation.

3. Example 1: Defining an Integrated View on Products

3.1. The unfeasibility of a single product catalogue

Product developers are responsible for the definition of new sellable products. Specific for the telecommunications business area is that the selling of a product is a long-lived transaction with the customer and that it involves a large number of configuration activities at the customer site. For example, the ordering of a point-to-point connection from site A to site B involves the installation of an unbundled line at both sites, the configuration of a router for each unbundled line and the configuration of a virtual circuit offering the required bandwidth.

Usually, an integrated approach tries to achieve the definition of a single product catalogue. However, people from different business area have a different view on products and have different needs. In an attempt to keep a unified view on products while accommodating for their different needs, people tend to twist product definitions in their respective software packages. By abusing attributes and fields for cross-referencing purposes, they try to maintain a more or less integrated approach. However, as the set of products in the product catalogue will increase, a unified view is no longer sustainable. Also in an international set-up with multiple business units a unified view can be held no longer: what is a single product from a sales point of view requires different technical configuration activities depending on the business unit. For example, Internet Access can be implemented by means of an unbundled line in the Netherlands and the UK, but must be provided with a leased line in Belgium, where unbundling of the local loop is not (yet) possible.

A scalable design must find a way to reconcile the differences between the technical and the commercial perspectives that can be taken on a product.

Since it must be possible for marketing people to define new products at any time, the product domain has been modelled by making use of the TypeObject Pattern, rather than by using generalisation/specialisation hierarchies. The TypeObject Pattern splits a class in a Type Class and an Instance Class and replaces subtypes of the original class by instances of the Type Class. For

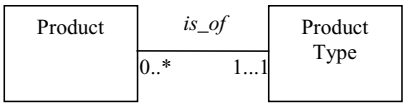

Fig. 3. *Type object pattern for products.*

products this means that rather than making a subclass of *Product* for each new type of product, it suffices to create an instance of *Product Type* that represents the new type of product (see Fig. 3).

The Product Domain is constituted entirely of Type Object Classes. Business rules were introduced to guard the mapping of the Type Object Classes with the Object Classes in the Order and in the Configuration Domain. As mentioned before, the Product Domain can be observed both from a commercial and a technical perspective. First the commercial perspective is presented.

3.2. Commercial perspective of the product domain

The formerly ad hoc composition of products in bundles, (sub)packages and options, is now formalised in the Enterprise Layer by means of a Product Composition Tree (depicted in Fig. 4). The leaf nodes of the product composition tree consist of installation orders, which are units of configuration. One level up, installation orders can be grouped into sub-packages. A grouping of sub-packages can constitute together a main package. Finally, at the highest level a number of main packages can be bundled together. Installation orders thus represent units of service provisioning and are the basic building blocks for marketing to compose new products.

The corresponding class diagram is given in Fig. 5. Customers can order an instance of a *PackageType*,

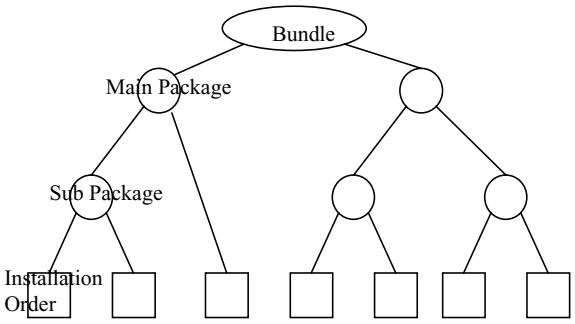

Fig. 4. *Product composition tree.*

in which case an object *PackageOrderLine* is created. By means of the association class *CompositionType*, the product composition tree can be modelled: *PackageType* is composed in a recursive way of other *Package Type*s (lower level packages). At the lowest level they are composed of elementary *InstallationOrderType*s. The following class constraint on object class *PackageType* prevents that a package could be composed of itself.

self.giveAllChildren.includes(self) = false
with
giveAllChildren: set[PackageType]
postcondition
result = union(self.BundlePT.SubPT.
giveAllChildren) ∪ {self}

By means of the association class *OptionType*, the options of a package are specified. Packages can be ordered as lower price options to other packages, resulting in the creation of objects of class *OptionOrderLine*. When a package is actually ordered as an option to a previously ordered package, a business rule has to verify whether the former is indeed modelled as an option to the latter in the corresponding Type Classes *OptionType* and *PackageType*. The following class constraint on *OptionOrderLine* enforces this business rule:

self.OptionType.MainPT
= self.PackageOrderLine.PackageType

3.3. Technical perspective of product domain

The technical perspective maintains for each type of installation order the parts to install and the parameters to configure. As depicted in Fig. 5, the association class *PartTypeUse* relates an *InstallationOrderType* with a *PartType*. A part type is an atomic installation unit, which can be described by a number of configuration parameters. An unbundled line, a router and a virtual circuit are all examples of part types. Some part types need the installation of other part types before they can be installed. For example, a virtual circuit requires the installation of two unbundled lines and an unbundled line requires in its turn the configuration of a router. To specify this hierarchical dependency the association class *PartPrerequisiteType* is introduced.

From these *Type* classes, a real configuration can be derived by instantiation of the classes *InstallationOrder*, *PartUse*, *Part* and *PartPrerequisite*. The usage relationship allows a *Part* to be (re)used by many

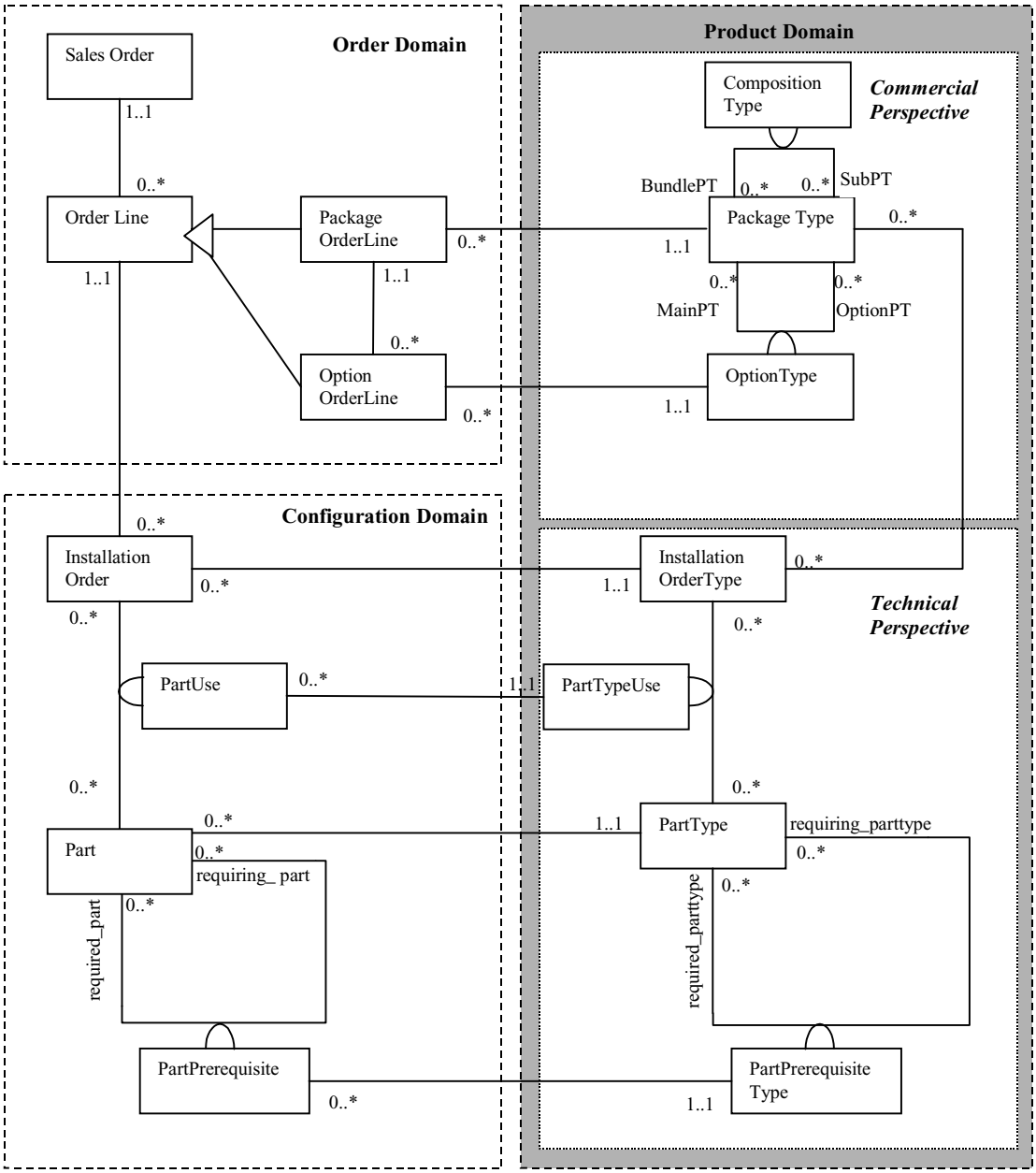

Fig. 5. *Class diagram covering the product, the order and the configuration domain.*

InstallationOrders. For example, a physical line can be used to implement many virtual circuits, e.g. one for Internet Access and one for a point-to-point connection to another customer site. The following class constraints on *PartUse* enforce that parts attached to an installation order during configuration activities, correspond to the configuration defined in the *Type* classes:

self.InstallationOrder.InstallationOrderType
= self.PartTypeUse.InstallationOrderType
self.Part.PartType = self.PartTypeUse.PartType

Finally, the Enterprise Layer has to ensure that the part prerequisite hierarchy imposed by object class

PartPrerequisiteType is accomplished during configuration activities. Therefore, the following class constraints on *PartPrerequisite* were included:

self.requiring_part.PartType
= self.PartPrerequisiteType.requiring_parttype
self.required_part.PartType
= self.PartPrerequisiteType.required_parttype

3.4. Positive effects on business processes support

The development of the Enterprise Layer and the resulting definition of the concept of PRODUCT have had a major impact on the product development process. A product has become more than the definition of hardware and how to configure it. In addition to the definition of hardware and configuration parameters, product developers must now fully elaborate the commercial perspective and the technical perspective on a product. By allowing two perspectives on products, the technical world and the commercial world have been separated. At the same time, the relationship between both worlds has been formalised. Product developers are now able to define the "translation" of a commercial product into a technical view by linking installation orders to packages. In this way, the structure of a package now clearly reflects the amount of work that is required to install a particular product. Since the introduction of this product structure, the company has observed a more transparent structuring of products: the existing product definitions have been restructured in order to better conform to the required installation steps.

The Enterprise Layer is also an important tool in passing information from one business area to another. The integrated approach makes information available for all applications and prevents the re-entering of the same data more than once. For example, information on customer configuration is built in the TBS MetaSolv application during service provisioning activities. This information must be available (at the right level of detail) for the Customer Support application. The Enterprise Layer and the respective co-ordination agents enable this. In addition, the Enterprise Layer also allows consolidating data that is present in different business areas, such as ordering information and configuration information for a single customer. For example, in the Enterprise Layer configuration information is directly linked to sales information. For each part at a customer site the corresponding installation orders can be traced. At their turn, installation orders are always related to an order line that gives the reason for the installation. In this way, the intermediary of installation orders translates order lines to activities on a customer configuration.

4. Example 2: Managing Sequences of Business Activities

4.1. Full processing of orders

Whereas the first example was in essence an example of information *structure*, the second example illustrates the behavioural perspective. In adopting company-wide business rules, the Enterprise Layer can also formalise existing work procedures. To illustrate this, the full process of ordering a product will be discussed (see Fig. 6).

A sales person has a lead, visits the customer, consults technical people for the optimal choice of products, reaches an agreement on what the customer wants to buy, registers the sales order, prints it and has it signed by the customer. The Sales Order is then passed on to the Service Provisioning Department. They register the sales order in their planning and install the products at the customer's site. At the end of the installation process, the customer signs for the acceptation of the products and from then on the billing of the installed services is started.

As can be seen from the description of the business process, activities of the different departments must be co-ordinated in a proper way. However, there is a total lack of integration between the billing application Geneva and the customer configuration application Metasolv. Integration between Metasolv and sales activities is also not formalised since there is no supporting software for Sales. As a result, human operators have to act as an intermediary, causing delays and inefficiencies in the transaction processing. The temptation to build ad hoc bridges between Geneva and Metasolv was substantial and has been reduced by building a provisional prototype of the Enterprise Layer in MsAccess (see Section 5 on prototyping).

The definition of the behavioural aspects of classes in the Enterprise Layer has allowed formalising the co-ordination of activities between the different departments. The *Type* classes in the Enterprise Layer define the relationship between order lines, installation orders, part uses and parts. When a customer orders a package, a new order line is created. In terms of enterprise modelling, this requires the following business events: *create SO* (Sales Order), *modify SO*, *create OL*

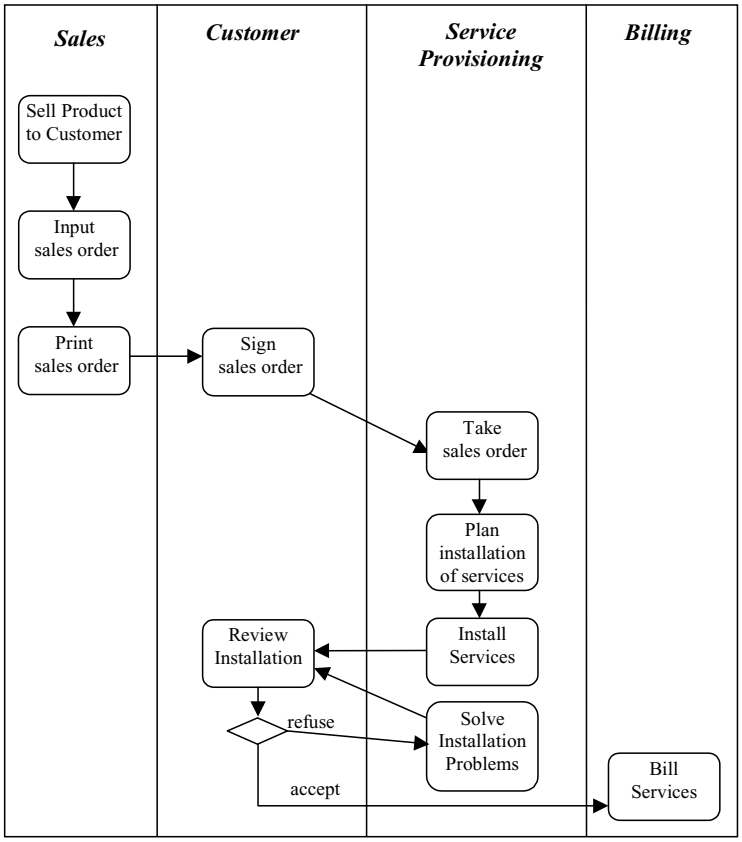

Fig. 6. *Full business process for ordering products.*

(Order Line), *modify OL*, and *end OL*. The *customer-sign* event models the fact that a final agreement with the customer has been reached (signature of sales order form by the customer). At the same time this event signals that installation activities can be started. In the same way, when all the parts of an installation order are configured, an event has to be generated signalling the completion of the installation order. The Enterprise Layer is a passive layer, and thus not capable of generating events. This is the responsibility of the user interfaces and of the so-called co-ordination agents build on top of the Enterprise Layer. They ensure by generating the appropriate events that state changes of the business classes are always reflected in the Enterprise Model.

The Finite State Machines of the object classes *SalesOrder*, *OrderLine* and *InstallationOrder* are now discussed. Also, the responsibility of the MetaSolv agent in generating the appropriate events will be indicated.

4.2. Finite state machine of SalesOrder (order domain)

A new sales order can be entered by means of a so-called Service Order Form (SOF) in the user interface (Fig. 7). As long as it is not signed, the sales order stays in the state "existing". The *customer_sign* event moves the sales order into the state "registered". From then on the sales order has the status of a contract with the customer and it cannot be modified any more. This means that the events *create OL*, *mod OL* and *end OL* are no longer possible for this sales order.

4.3. Finite state machine of OrderLine (order domain)

By listening to the *customer_sign* event, the MetaSolv agent knows when a sales order is ready for processing for Service Provisioning. In accordance with the *Type* business classes, the MetaSolv agent creates for each order line on the sales order, the corresponding installation orders. Now, Service Provisioning has to check

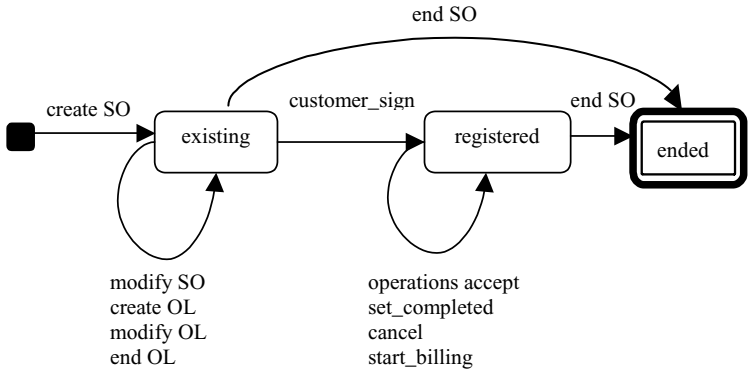

Fig. 7. *State machine for sales order.*

whether these installation orders are configurable. If so, an *operations_accept* event is generated for the order line, altering the order line from state "existing" to state "in progress" (see Fig. 8). The installation orders can now be configured. Meanwhile, the MetaSolv agent tries to generate a *set_complete* event for the order line. However, a business rule in the Enterprise Layer will prevent the completion of an order line if not all the corresponding installation orders are completed:

```
Class OrderLine
...
set_complete (...) is
        Precondition
                ∀ Self.InstallationOrder :
                  InstallationOrder.state
                  = completed"...
        do
        ...
        end
```

4.4. Finite state machine of InstallationOrder (configuration domain)

By creating an installation order, it enters the state "existing": the corresponding part uses can be created and configured (Fig. 9). When an installation order is registered as completed in the Metasolv software, the MetaSolv agent sends a *complete IO* event to the Enterprise Layer, thereby altering the state of the installation order from "existing" to "completed".

4.5. Effects on the business processes

Before the Enterprise Modelling effort, the sales process was not supported by a BSS/OSS and it was mostly paper-based. Initially, it was assumed that a sales order would be registered in the Enterprise Layer once agreement has been reached with the customer. The first part of the business process is then still a paper-based process until the paper SOF has been signed. Registered sales orders are immediately passed on to Customer Operations. However, it frequently happens that

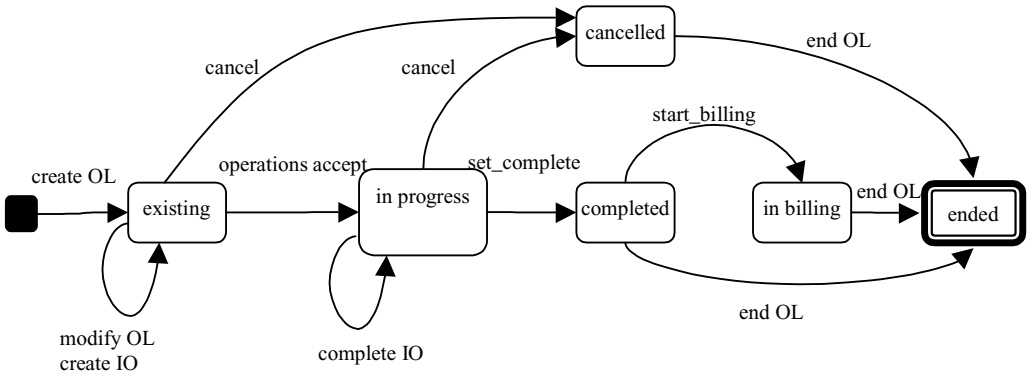

Fig. 8. *Finite state machine of orderline.*

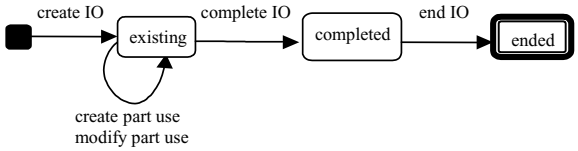

Fig. 9. *Finite state machine for installation order.*

customers change their mind. In the past, sales orders were changed even when installation was already in progress. This was however only possible for "minor" changes, such as a change of line speed from 256 kb to 512 kb, because this only requires a different configuration of an installed part. "Major" changes, such as those that require the installation of other parts, were avoided as much as possible. Whether or not the customer had to sign a new SOF was not clear. As it turned out to be too complex to formalise the difference between "minor" and "major" changes, the new automated business process forbids the modification of a sales order once it has been accepted by Customer Operations. Since not all orders in the state "modifiable" can be accepted by Customer Operations, but only those that are "ready", an additional business event type *customer_sign* was introduced that "freezes" a sales order. Modifications are forbidden after the *customer_sign* event. If a modification is required, this must be registered as a new order that replaces the old one. The effect on the technical side will be different whether installation orders are completed or not.

In analogy with the modelling of the structural aspects, also the modelling of behavioural aspects of domain objects requires well-defined business processes. And also during this phase of the enterprise-modelling project, the relationship between different business areas had to be clarified. For example, the Enterprise Layer now allows monitoring signals from the Service Provisioning area (such as the completion of installation orders) and using these signals to steer the completion of sales orders. At its turn, the billing agent will use the completion of a sales order to start the billing process. In this way, the Enterprise Layer allows for an automated co-ordination of behavioural aspects of different business areas.

5. Prototyping

In the Enterprise Layer, products are defined as package types with a tree-structure. In order to gain more confidence with the dual view on products (commercial versus technical), paper and blackboard simulations of possible uses of the Enterprise Layer were made. Later on, a provisional version of the Enterprise Layer was built in MsAccess.

As the Enterprise Layer takes more than one-year time to be specified and developed, there was a lot of pressure to develop ad-hoc bridges between the existing BSS/OSS. People don't want to enter the same information more than once. The prototype has released some of this pressure. Sales information is now entered in the provisional Enterprise Layer and human co-ordination agents copy information manually to the BSS/OSS in a well-structured way.

During the prototyping of the Enterprise Layer it appeared that current products have only 2 levels: the level of installation order type and one package type level (see Fig. 4). Hence, at present the product composition tree offers more possibilities than required. It was however decided to keep the tree structure to have sufficient degrees of freedom for the definition of future products.

Prototyping also revealed difficulties in usage of the Enterprise Layer. Product developers need to build some expertise in deciding what to group in a single installation order type and what to split into several installation order types. Also the rules for deciding what to consider as an option of a package or as a different package have to be elaborated by experience. For example, Internet Access can be offered at different speeds (256 KBit, 512 KBit,...). Is speed an option of a single product "Internet Access" or is it better to consider Internet Access 256 as a first product and Internet Access 512 as a second product? Prototyping has allowed defining guidelines that help people use the Enterprise Layer in the best possible way.

Another goal of the prototyping effort was the prevention of the registration of too much badly structured information in the existing applications. Prototyping allowed people to register information in the right way in the existing applications, what substantially reduces conversion problems when the final Enterprise Layer is taken into production.

6. Conclusions

The paper discussed the specification of an Enterprise Layer for a young company starting in the telecommunications sector. This specification was a non-trivial

project for two main reasons. On the one hand, as Novaxess is a new company building everything from scratch, work procedures were not always precisely defined: in practice many different scenarios were applied in very similar situations. Since there is a strong interplay between the domain model and the work procedures, this forced us to constantly monitor the effect of the particular choices in the design of the Enterprise Layer on work procedures. Defining which procedure to use in what situation, and formalising the appropriate business rules was a time consuming task.

On the other hand, the installed BSS/OSS are tailored to their specific business area. Hence, they have very specific and different views on the domain, both from a structural and a behavioural perspective. Each software package had its own particular definitions of concepts and ways of working. Defining a generic view that can accommodate for the specific needs of each business area was a major challenge.

The project has lead to some useful lessons in domain modelling. For a full coverage description of the business at least three dimensions can be identified: the *intentional dimension* describing goals and strategy, the *dimension of the domain model* modelling all relevant domain concepts and the *organisational dimension* treating the different ways of working (Nellborn, 1999; Nilsson, 1999). The method that was used (MERODE) is a domain modelling method with no support for business process modelling. But the same remark holds for other well-known object-oriented analysis methods (Booch, Rumbaugh, and Jacobson, 1999; Coleman et al., 1994; Cook and Daniels, 1994; D'Souza and Wills, 1999)[5] . Although Use cases and Activity Diagrams are sometimes advocated as more process oriented techniques, they are not meant nor adequate for business process modelling (Snoeck, Poelmans, and Dedene, 2000).

A positive aspect of the method is the recognition of business events as independent concepts (which is a particular feature of MERODE). This facilitates the link between the business process model and the Enterprise Model. A future project will define business processes in terms of sequences of business events and develop monitoring facilities that allow following up the progress of a given process. A classical approach where behaviour is modelled as methods of domain object classes and interaction is modelled by means of message passing would have made such a link more difficult. In addition it would also have made a clear separation between business process aspects and domain model aspects more difficult (Lindström, 1999).

Domain modelling is definitely an iterative process. The first approach was an area-by-area analysis, but soon it became clear that the co-ordination of these areas has a substantial influence on the overall domain structure. As a result, an enterprise-wide approach with iterative refinements is to be advocated as the best approach. An in-depth study of only one business area, followed by the in-depth study of the next-area would lead over and over to major changes of the previously obtained results.

The definition of the behaviour of domain object types is strongly related to the business process aspects. Designing the behaviour of domain classes cannot be done in isolation: co-design or at least analysis of the low-level business processes is a necessity. One of the difficulties with this is that it is not always clear when sequence constraints on business events result from essential business rules and when they are (only) the result of a particular way of working. A well-done separation of business process aspects versus domain aspects is however essential for the construction of flexible, adaptable systems. More research is required to find modelling guidelines to assist analysts in achieving the "perfect" separation of concerns.

A final remark is that there are left a number of degrees of freedom in the Enterprise Model. Business rules that would pose too strong restrictions on business processes were not included in the model, and some object classes were introduced that are currently redundant but could be useful in the future. The model developed so far is also extensible, meaning that next versions will be evolutions and not complete replacements of the current model.

Enterprise Modelling has helped a lot in the discovery of business rules. These experiences confirm other people's findings that the gap between the perceived business realities versus what can be represented in a computerised way is usually wider and deeper than initially thought. Business modelling is a way to bridge that gap and at the same time it helps constructing a better understanding of what the business really is like (Lindström, 1999). In addition, the automated world has more possibilities than a manual world. As a consequence it is not sufficient to automate what people would do manually. Work procedures that are much too cumbersome for people are perfectly feasible in an automated world (computers never complain). Moreover, Enterprise Modelling is a way to achieve good solutions

that go beyond the day-to-day problems. The elaboration of business rules raises questions that would not be considered if the focus was only on solving today's problems. Finally, these experiences also confirm that the use of an object-oriented modelling approach is an important factor for the success of the project (Galfione et al., 2000): the analysis of the behavioural aspects of domain classes raised a lot of questions that would have remained unanswered in case of a purely structural approach.

Acknowledgment

The work described in this paper is the result from a project executed with the company NOVAXESS based in Amsterdam, the Netherlands.

Notes

1. MetaSolv®. Available at http://www.MetaSolv.com (MetaSolv® is a registered trademark, www.metasolv.com).
2. Geneva. Available at http://www.genevatechnology.com/.
3. Clarify. Available at http://www.nortelnetworks.com/products/04/cefo/.
4. Johnson Ralph. Dynamic Object Model, *ObjectiveView*, Issue 5, Ratio, available at www.ratio.co.uk.
5. Rational Software Corporation. The Unified Modelling Language. Available at http://www.rational.com/.

References

Booch G, Rumbaugh J, Jacobson I. *The Unified Modeling Language User Guide*. Reading, MA: Addison Wesley, 1999.

Coleman D, et al. *Object-oriented development: The FUSION method*, Prentice Hall, 1994.

Cook S, Daniels J. *Designing Object Systems: Object-Oriented Modeling with Syntropy*, New York: Prentice Hall, 1994.

D'Souza DF, Wills AC. *Objects, Components and Frameworks with UML, The Catalysis Approach*. Reading, MA: Addison-Wesley, 1999:785.

Galfione P, Galdiolo A, Valerio A, Cardino G. Exploiting enterprise knowledge through domain analysis and frameworks: An experimental work. *Proceedings of the Eleventh International Workshop on Database and Expert Systems Application*, DomE 2000, 4–8 September, Greenwich, London, UK, IEEE Computer Societey, 2000:813–822.

Lindström C. Lessons learned from applying business modelling: Exploring opportunities and avoiding pitfalls. In: Nilsson AG, Tolis C, Nellborn C, eds., *Perspectives on Business Modelling, Understanding and Changing Organisations*, Berlin: Springer-Verlag, 1999.

Nellborn C. Business and systems development: Opportunities for an integrated way of working. In: Nilsson AG, Tolis C, Nellborn C, eds., *Perspectives on Business Modelling, Understanding and Changing Organisations*, Berlin: Springer-Verlag, 1999.

Nilsson AG. The business developer's toolbox: Chains and alliances between established methods. In: Nilsson AG, Tolis C, Nellborn C, eds., *Perspectives on Business Modelling, Understanding and Changing Organisations*, Berlin: Springer-Verlag, 1999.

Snoeck M, Dedene G. Existence Dependency: The key to semantic integrity between structural and behavioral aspects of object types. *IEEE Transactions on Software Engineering* 1998;24:233–251.

Snoeck M, Dedene G, Verhelst M, Depuydt AM. *Object-Oriented Enterprise Modeling with MERODE*. Leuven University Press, 1999.

Snoeck M, Poelmans S, Dedene G. A layered software specification architecture. In: Laendler AHF, Liddle SW, Storey VC, eds., *Conceptual Modeling-ER2000, 19th International Conference on Conceptual Modeling*, Salt Lake City, UTAH, USA, Lecture Notes In Computer Science, Vol. 1920, Berlin: Springer-Verlag, 2000:454–469.

Monique Snoeck obtained her Ph.D. in May 1995 from Computer Science Department of the Katholieke Universiteit Leuven with a thesis that lays the formal foundations of MERODE. Since then she has done further research in the area of formal methods for object-oriented conceptual modelling. She now is Associate Professor with the Management Information Systems Group of the Department of Applied Economic Sciences at the Katholieke Universiteit Leuven in Belgium. In addition, she is invited lecturer at the Université Catholique de Louvain-la-Neuve since 1997.

She is and has been involved in several industrial conceptual modeling projects. Her research interest are object oriented modelling, software architecture and gender aspects of ICT.

Cindy Michiels is a Ph.D. student at the Katholieke Universiteit Leuven (KUL), Belgium. She holds a bachelor degree in Applied Economics and a master degree in Management Informatics (2000). Since 2001 she is working at the Management Information Systems Group of the department of Applied Economics of the KUL as a doctoral researcher. Her research interests are related to domain modelling, software analysis and design, and automatic code generation.